Multi-Target Vectorization With MTPS C++ Generic Library

Wilfried Kirschenmann[1,3], Laurent Plagne[1], and Stéphane Vialle[2,3]

[1] SINETICS Department, EDF R&D, FRANCE,
{wilfried.kirschenmann,laurent.plagne}@edf.fr
[2] SUPELEC - UMI 2958, FRANCE,
stephane.vialle@supelec.fr
[3] AlGorille INRIA Project Team, FRANCE

Abstract. This article introduces a C++ template library dedicated at vectorizing algorithms for different target architectures: Multi-Target Parallel Skeleton (MTPS). Skeletons describing the data structures and algorithms are provided and allow MTPS to generate a code with optimized memory access patterns for the choosen architecture. MTPS currently supports x86-64 multicore CPUs and CUDA enabled GPUs. On these architectures, performances close to hardware limits are observed.

Keywords: GPU, SSE, Vectorization, C++ Template Metaprogramming, Performances

1 Introduction

In many scientific applications, computation time is a strong constraint. Optimizing these applications for the rapidly changing computer hardware is a very expensive and time consuming task. Emerging hybrid architectures tend to make this process even more complex.

The classical way to ease this optimization process is to build applications on top of High Performance Computing (HPC) libraries that are available for a large variety of hardware architectures. Such scientific applications, whose computing time is mostly consumed within such HPC library subroutines, then automatically exhibit optimal performances for various hardware architectures.

However, most classical HPC libraries implement fixed APIs like BLAS which may make them too rigid to match the needs of all client applications. In particular, classical APIs are limited to manipulate rather simple data structures like dense linear algebra matrices. As a more complex issue, general sparse matrices cannot be represented with a unified data structure and various formats are proposed by more specialized libraries. In the extreme case, structured sparse matrices cannot be efficiently captured by any of the classical library data structures. Several neutron transport codes developed at EDF R&D rely on such complex matrices that another kind of library is required.

Following the model of the C++ Standard Template Library (STL), template based *generic libraries* such as Blitz++ [13] provide more flexible APIs and

extend the scope of library-based design for scientific applications. Such generic libraries allow to define Domain Specific Embedded Languages (DSELs) [2].

Legolas++ (*GLASS* in [11]), a basis for several HPC codes at EDF, is a C++ DSEL dedicated to structured sparse linear algebra. In order to meet EDF's industrial quality standards, a *multi-target* version of Legolas++, currently under development, will provide a unified interface for the different target architectures available at EDF, including clusters of heterogeneous nodes (i.e., with both multi-core CPUs and GPUs). This article presents MTPS (Multi-Target Parallel Skeletons), a C++ generic library dedicated to *multi-target* vectorization that is used to write the *multi-target* version of Legolas++. Only developments concerning a single heterogeneous node are presented here.

The next section presents the principles of Legolas++ and Section 3 introduces MTPS. Its optimization strategies and the achieved performances are discussed in Section 4. Finally, conclusions are drawn in Section 5.

2 Towards a Multi-Target Linear Algebra Library

Legolas++ is a C++ DSEL developed at EDF R&D to build structured sparse linear algebra solvers. Legolas++ provides building bricks to describe structured sparse matrix patterns and the associated vectors and algorithms. This library separates the actual implementation of the Linear Algebra (LA) computationnal kernels from the physics issues.

Fig. 1. Block Matrix Patterns. A block diagonal matrix pattern is represented on the left while a block diagonal matrix with tridiagonal blocks is represented on the right.

Block decomposition is a common linear algebra operation that allows to describe the sparsity pattern of a given matrix from one or several basic sparsity patterns. For example **Fig. 1**(left) represents a matrix with a block diagonal sparsity pattern that can help to identify the optimal algorithm for dealing with this matrix. If a matrix block can itself be block-decomposed, the matrix is said to be a multi-level block matrix. **Fig. 1**(right) represents such a multi-level block matrix which is diagonal with tridiagonal blocks. Legolas++ is a C++ library developed at EDF R&D for structured sparse linear algebra problems. The central issue in this domain is to describe efficiently multi-level block matrices as combinations of basic sparsity patterns. Legolas++ allows to access the block elements of a block matrix in the same manner as if it was a simple matrix. For

example `A(i,j)` returns a reference to the `(i,j)` matrix element which can be either a scalar if `A` is a matrix, or a block if `A` is a block matrix. In the latter case, this block can be seen as a simple sub-matrix and provides the same interface. This means that `A(i,j)(k,l)` returns a reference to the `(k,l)` scalar element of the `(i,j)` block.

Such a block matrix naturally operates with 2D vectors. For example let us consider the following matrix-vector product $Y = Y + A * X$ where A is a block matrix and `X` and `Y` are vectors. Legolas++ allows to implement this product as:

```
1 for (int i=0 ; i < A.nrows() ; i++)
2  for (int j=0 ; j < A.ncols() ; j++)
3   Y[i]+=A(i,j)*X[j];
```

In this case, each elementary operation `Y[i]+=A(i,j)*X[j]` corresponds to a simple matrix-vector sub-product and the C++ compiler statically transforms the previous handwritten algorithm into the following generated block algorithm:

```
1 for (int i=0 ; i < A.nrows() ; i++)
2  for (int j=0 ; j < A.ncols() ; j++)
3   for (int k=0 ; k < A(i,j).nrows() ; k++)
4    for (int l=0 ; l < A(i,j).ncols() ; l++)
5     Y[i][k]+=A(i,j)(k,l)*X[j][l];
```

The previous algorithm shows that the vectors `X` and `Y`, corresponding to the block matrix `A`, are two-dimensional. In order to simplify the Legolas++ vocabulary, one describes a block matrix like `A` as a 2-level Legolas++ matrix that operates on 2D Legolas++ vectors. The main objective of the Legolas++ library is to provide tools for the users to explicitely define their n-level matrices and corresponding nD vectors. For instance, **Fig. 2** shows the sparsity pattern of a 5-level Legolas++ matrix block that belongs to the 7-level matrix of our neutron transport code [11,6,8].

The explicit GPU parallelization of a neutron transport code resulted in speed-ups around 30 over the sequential Legolas++ CPU implementation [6,8]. To generalize this gain of performances to other Legolas++ based applications, a parallel and multi-target version of Legolas++ is to be developed. As the parallel CPU and GPU versions exhibit strong similarities, Legolas++ developments for

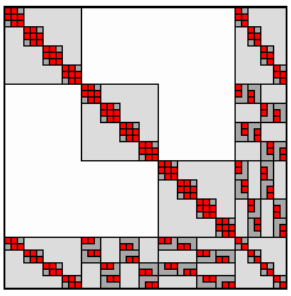

Fig. 2. Sparsity Pattern of a 5-level Legolas++ block matrix.

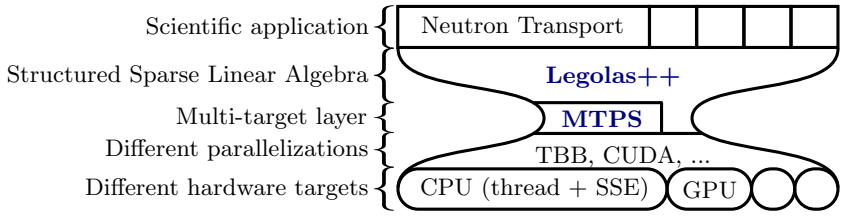

Fig. 3. Our hourglass software architecture to achieve a multi-target Legolas++: a minimal MTPS library adapts the code for different hardware architectures.

the different targets are factorized into an intermediate layer between Legolas++ and the different hardware architectures, namely MTPS (see **Fig. 3**). Note that examples provided in the following of this paper correspond only to MTPS code as the multi-target version of Legolas++ is currently under development.

3 Introduction to MTPS

3.1 Related Work

Many libraries parallelize for different architectures from a single source code. A complete bibliography is beyond the scope of this article; only some examples based on C++ meta-programming techniques are introduced.

Some libraries, like TrilinosNode [1], Quaff [4] or Intel TBB [12], require their users to explicitly express the parallelism within the application by using *parallel skeletons*. This expression of available parallelism can be encapsulated into specialized and implicitly parallel STL-like *containers* and *algorithms*, as in Thrust[4] and Honei [3].

Our goal is to provide implicit parallelism within Legolas++ *containers* and *algorithms*. To ease the writing of its *containers* and *algorithms*, Legolas++ relies on MTPS which follows a *parallel skeletons* based approach. Then MTPS optimizes the code for the different architectures.

As this article presents MTPS, only code using MTPS is shown. However Legolas++ will hide MTPS details in its *containers* and *algorithms* so its user do not need to be aware of MTPS.

3.2 *Collections* and *Vectorizable Algorithms*

This section introduces the notions of *collection* and *vectorizable algorithm* on which MTPS relies.

In C++, a *Plain Old Data* (POD) is a data structure that is represented only as passive collections of field values, without using encapsulation or other object-oriented features. POD non-static data members can only be integral types or PODs. As a POD have neither constructor nor desctructor, it can be copied

[4] Thrust: `http://code.google.com/p/thrust/`

or moved in memory [5]. This particularity allow MTPS to copy a POD from one memory space to another (e.g., GPU memory space). In the following code snippet, `MyPOD` is a POD with three `float` data members:

```
1 struct MyPOD{ float a,b,c; };
```

Let a *collection* be a data structure containing different instances of the same POD and `f` be a *pure* function (i.e., `f` has no side effects). An algorithm applying `f` to all elements of a *collection* is said to be *vectorizable*. To parallelize such algorithms, MTPS provides two parallel skeletons optimized for different target architectures: *map* and *fold* which correspond to a *parallel for loop* and to a *parallel reduction* respectively.

An algorithm applied to a set of data is *vectorizable* if and only if this set of data is considered as a *collection* and if the algorithm can be decomposed as a *pure* function applied to each element of the *collection*. We say that an algorithm is *vectorizable* in reference to a given *collection*. For instance, an algorithm operating on each row of a matrix is *vectorizable* only if the matrix is considered as a *collection* of rows. The same algorithm is not *vectorizable* if the matrix is considered as a *collection* of columns: the matrix must be transformed (i.e., transposed). Two algorithms *vectorizable* in reference to the same *collection* are said to be in the same *vectorial context*. On the contrary, if two consecutive algorithms are not *vectorizable* in reference to the same *collection*, a *context switch* (the matrix transposition in our example) is required. In a distributed memory system, *context switches* correspond to communications.

3.3 Linear Algebra *Hello World* of MTPS: `saxpy`

This section presents how to use MTPS to implement the `saxpy` operation and to execute it efficiently on different target architectures. The `saxpy` operation is part of the BLAS interface and its C implementation is:

```
1 float *X, *Y, a;
2 for(int i=0; i<N; ++i) Y[i]+=a*X[i];
```

First, the iteration-dependent data are gathered in a POD `XYData` whose members correspond to `X[i]` and `Y[i]`. The types of the two members (`float`) are passed as template arguments to `MTPS::POD` and their names (`x` and `y`) are given in the `Fields` enum:

```
1 struct XYData: public MTPS::POD<float,float>{
2  enum Fields{x, y};
3 };
```

Second, a *collection* of `XYData` elements, `xyCol`, is built using MTPS containers. Optimized containers are provided as member of the class corresponding to the target architecture. Two levels of parallelism are available on CPUs: thread parallelism and SIMD parallelism. The choice for each level is made by passing two arguments to the `CPU` template class. `Thread` can be one of `MTPS::Sequential`, `MTPS::OMP` (openMP) or `MTPS::TBB` (Intel TBB). `SIMD` can

be one of `MTPS::Scalar` (no SIMD instruction generated) or `MTPS::SSE` (SIMD instruction generated using SSE intrinsics). On CUDA-enabled GPUs, only the SIMD parallelism is provided.

```
1 //typedef MTPS::GPU::CUDA Target;        // To use the GPU
2
3 //typedef MTPS::Sequential Thread;       // Single threaded
4 //typedef MTPS::TBB Thread;              // TBB parallelism
5 typedef MTPS::OMP Thread;                // OpenMP parallelism
6 //typedef MTPS::Scalar SIMD;             // Disable SIMD units
7 typedef MTPS::SSE SIMD;                  // Enable SSE units
8 typedef MTPS::CPU<Thread, SIMD> Target; // To use the CPU
9
10 Target::collection<XYData> xyCol(N);
```

Third, the function that is to be applied to all elements of the collection must be written as a functor class: `AxpyOp`. The coefficient `a` is common for all elements of the collection and is stored as a member of the `AxpyOp` functor class:

```
1 struct AxpyOp{
2  float a_;
3  template <template <class> class View>
4    INLINE void operator()(View<XYData> xy) const {
5   typedef View<XYData> XYV;
6   int x = XYV::x;
7   xy(XYV::y)+=a_*xy(x);
8  }
9 };
```

As `XYData` elements may not be stored identically on different target architecture, `AxpyOp::operator()` does not take an `XYData` as argument. A `View` is provided instead. `XYData` members can be accessed with the `operator()` of the `View` which takes an `int` as argument. This `int` identifiy the data member that is to be accessed; either `X[i]` or `Y[i]` in our exemple. Elements of the `Fields` enum can be used either to initialize an `int` (line 7) or directly (line 8). The declaration of `AxpyOp::operator()` must be preceeded by the `INLINE` macro which defines target-dependent keywords (e.g. `__device__` for CUDA).

Finally, the functor is passed to the `map` and `fold` parallel skeletons provided by the *collection* container:

```
1 AxpyOp axpyOp; axpyOp.a_=...;
2 xyCol.map(axpyOp);
3 ...
4 DotOp dotOp;
5 float dot = xyCol.fold(dotOp);
```

Although more verbose and harder to use than the approaches presented in Section 3.1, this formalism allow MTPS to be the only library at our knowledge that optimizes the data layout for different architectures as Section 4 will show.

4 Optimization of Performances

For each architecture, the specific optimizations required to enable good performances will be presented. The implementation of a more complex example will then be discussed.

4.1 Multi-Target Performance Optimizations

Parallelizing a *vectorizable algorithm* is straightforward. However, achieving good performances on different hardware architectures is not: modifications of the *collection* data structure may be required. Indeed, achieving efficient usage of memory bandwidth on a given hardware architecture requires specific access patterns [7]. **Fig. 4** shows a block-diagonal matrix of 8 TriDiagonal Symmetric Matrix blocks (TDSM) of size 4 (left). Assuming that this matrix is considered as a *collection* of TDSM blocks in reference to an algorithm, **Fig. 4** shows how to store it on three different architectures to optimize the access pattern (right):

- on CPU (top), maximizing data locality is required to avoid cache misses. As the TDSM blocks are independent, data locality only matters inside a TDSM block. Hence, the best performances are achieved when each TDSM block is stored in a contiguous chunk of memory;
- on GPU (bottom), memory accesses have to be *coalesced* to achieve good performances (see the CUDA programming guide [10]). This implies that the accesses made by two threads i and $i+1$ must correspond to two elements at index j and $j+1$. As the same function is applied in a SIMD fashion to the different TDSM blocks, all elements A are accessed at the same time and they have to be stored in a contiguous chunk of memory;
- using the SSE units (middle) requires to pack the data into *vectors* containing 4 independent elements that have to be accessed together. Although a GPU ordering would fill this need, this would break the data locality and imply CPU cache misses. Finally, an intermediate storage between the two previous is optimal.

Performances achieved thanks to this optimization will be shown in Section 4.3. This optimization is made in MTPS *collection* container. To construct a *collection*, MTPS user must define both the size per POD-element of each field (4 for the diagonal field on **Fig. 4**) and the number of POD-elements. Using this information, MTPS optimizes the storage for each target architecture.

As a *context* corresponds to a storage pattern, a *context* switch imply a data reordering. For instance, switching a *collection* of matrix rows to a *collection* of matrix columns modifies the effective storage (i.e. the matrix is transposed). MTPS provides some switch skeletons.

4.2 Implementation of a Linear System Resolution

The example presented in this section corresponds to a basic operation that represents the major part of the execution time of a neutron transport code [6,8].

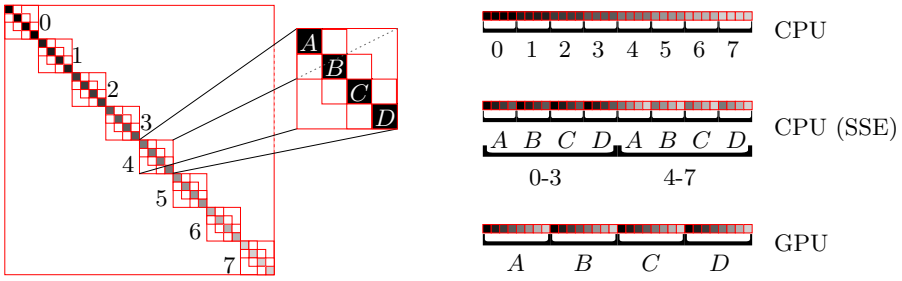

Fig. 4. The storage of the diagonal is adapted by MTPS for the target architecture.

Let $\mathbf{A}$ be a block-diagonal matrix with TDSM blocks. The $\mathbf{AX} = \mathbf{B}$ linear system can be seen as a *collection* of smaller block systems $ax = b$ that can be solved independently. To solve one $ax = b$ system, the matrix a is factorized in-place with a $\mathrm{LDL}^{\mathrm{T}}$ decomposition and a forward and backward substitution is then applied on x. Only the code for the factorization is shown here.

First, let us introduce `TData` which represents a TDSM block. `TData` elements are stored in two vectors corresponding to the diagonal and the lower diagonal:

```
1 struct TData: public MTPS::POD<float,float>{
2  enum Fields{diag, low};
3  typedef typename MTPS::POD<float,float>::Shape Shape;
4  static Shape createShape(int size){
5   Shape out;
6   out[diag] = size;       out[low] = size-1;
7  }
8 };
```

The `Shape` type of line 5 contains the effective sizes of the two fields. All elements of a *collection* have the same shape.

Second, to build a *collection* of `TData`, one has to provide both the number of `TData` elements and their shape. With these elements, the storage pattern of `tCol` can be optimized according to the target architecture (see **Fig. 4**):

```
1 TData::Shape s=TData::createShape(size);
2 Target::collection<TData> tCol(N, s);
```

Third, the `TLDLtOp` functor class that factorizes the matrix a in-place using a $\mathrm{LDL}^{\mathrm{T}}$ decomposition has to be provided:

```
1 struct TLDLtOp{
2  template <template <class> class View>
3   INLINE void operator()(View<TData> a) const{
4   typedef View<TData> TV;
5   int low = TV::low, diag = TV::diag;
6   typename TV::template Type<low>::Type low_i_1;
7   int size = a.shape()[diag];
8   for (int i = 1 ; i < size ; i++){
9    a(low, i-1)=a(low, i-1)/a(diag, i-1);
```

```
10    a(diag, i)-=a(diag, i-1)*a(low, i-1)*a(low, i-1);
11   }
12  }
13 };
14 TLDLtOp op;
15 tCol.map(op);
```

The elements of a field are accessed by passing their index as the second argument of the view `operator()`. If no index is provided as in the line 8 of the `AxpyOp` example of Section 3.3, the first element is returned. Line 7 shows how the type of the field elements can be retrieved.

4.3 Performances

Table 1 shows the performances obtained to compute the solution of the $\mathbf{AX} = \mathbf{B}$ system from Section 4.2 with $\mathbf{A}$ having 10^5 blocks of size 100×100. The matrix and vector are directly constructed on the target architecture and do not require further reordering to fit the target architecture. Speed-ups take the sequential scalar CPU version as reference. CPU tests are run on a machine with two 2.933 GHz Intel X5670 hexa-core processors. GPU tests are run on a Nvidia Quadro C2050 card. Both architectures were launched in 2010. On CPU, icpc 11.1 and g++ 4.5 provide the same performances. On GPU, nvcc 3.2 has been used. Computation performances are given in GFlops. Data throughput is given in GB/s and takes into account the effective data transfers to and from the memory. Consequently, an element remaining in the cache memory between two loads is considered to have been loaded only once.

For each architecture, the achieved performances are compared to the expected performances corresponding to the best observed performances on the given architecture. Expected computational power are measured with large BLAS matrix-matrix multiplications (`sgemm`): 11.2 GFlops with one CPU core and 126.5 GFlops with the 12 CPU cores using Intel MKL. The MKL uses the SSE units. As these units can execute 4 single precision floating point operations, we define expected computational performances without the SSE units as ¼ of the SSE performances (i.e., 2.8 GFlops and 31.6 GFlops respectively). On GPU, the expected computational power measured is 435 GFlops. On CPU, the expected memory throughputs are measured with an extended version of the stream benchmark [9]. This version adds a new subroutine containing 9 memory accesses (instead of 3 for the `Triad` routine) and shows 12.4 GB/s for single threaded execution and 35.0 GB/s for the parallel execution using openMP. On GPU, the expected memory throughput is measured with the CUDA SDK bandwidth benchmark on GPU: 86.3 GB/s.

To provide comparable results in spite of the hardware differences, the specifications of the hardware have been taken into account. For the computational power, the difference relies in the number of cycles to evaluate a floating point division: on GPU, 1 cycle is required but 15 are required on CPU. Thus, on CPU, the division is considered as 15 floating point operations. For the memory

throughput, only accesses to the main memory are counted. In other words, accesses to a piece of data that have already been loaded in cache are considered as free. On CPU, 3 accesses are saved whereas no accesses are saved on GPU. Evaluating the computational power and the memory throughputs this way allow us to make fair comparison to the expected performances.

The performances of a code on a given architecture are limited either by the computational power or by the memory throughput. Bold figures in **Table 1** correspond to the limiting factor for the corresponding target architecture. On CPU, when no or few parallelism is used, the performances are limited by the computational power: the performances achieved are between 75% and 93% of the expected computational performances. When both the threading parallelism and the SIMD parallelism are enabled, the performances are limited by the memory throughput: almost 100% of the best observed throughput is obtained. On GPU, all the available parallelism is used and the performances are thus limited by the memory throughput: 95% of the best observed throughput is reached.

Table 1. Performances of MTPS for the TDSM example. Computation are carried out in single precision floating point.

Thread	SIMD	Time	Speed	Computational Power			Data Throughput		
		(ms)	Up	GFlops	Expected	%Exp.	GB/s	Expected	%Exp.
sequential	scalar	131.9	1.0	2.5	2.8	**88**	1.7	12.4	14
	SSE	37.3	3.5	8.7	11.2	**78**	6.0	12.4	48
intel TBB	scalar	12.1	11.	27.0	31.6	**85**	18.5	35.0	53
	SSE	6.6	20.	49.4	126.5	39	33.9	35.0	**97**
openMP	scalar	11.1	12.	29.4	31.6	**93**	20.1	35.0	57
	SSE	6.5	20.	50.1	126.5	40	34.4	35.0	**98**
CUDA C		4.1	32.	15.9	435.	4	81.8	86.3	**95**

The limitation of the performances by the memory throughput shows the importance of optimizing the memory accesses. Finally, **Table 1** shows that by abstracting the memory access pattern to the target architecture the the performances of a given code can near the hardware limits on different architectures.

5 Conclusions and Perspectives

We have presented MTPS, a C++ generic library simplifying the parallelization and the optimization of *vectorizable algorithms* for different architectures. Although MTPS semantics and syntax remain complex, the end user should not be aware of this complexity: MTPS is designed to be generated, especiallly with the C++ template metaprogramming approach. Finaly, an algorithm written once with MTPS can be compiled to be executed on the SSE units of multicore CPUs or on CUDA-enabled GPUs and obtain performances close to hardware limits: more than 95% of peak performances were observed.

For further developments of MTPS, the design of an new version of Legolas++ on top of MTPS will allow to validate the set of skeletons provided by MTPS, especially concerning the *context switches*. The implementation of a neutron transport solver [6,8] with this version of Legolas++ will automaticaly provide a multi-target version of this solver. Efforts will be made to keep the portability of the performances currently available with MTPS.

Acknowledgement: authors want to thank Region Lorraine and ANRT for supporting this research.

References

1. Baker, C.G., Carter Edwards, H., Heroux, M.A., Williams, A.B.: A light-weight API for Portable Multicore Programming. In: PDP 2010: Proceedings of The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing. IEEE Computer Society, Washington, DC, USA (2010)
2. Czarnecki, K., Odonnell, J.T., Striegnitz, J., Walid, Taha: DSL Implementation in MetaOCaml, Template Haskell, and C++. LNCS: Domain-Specific Program Generation 3016(2), 51–72 (2004)
3. Dyk, D.V., Geveler, M., Mallach, S., Ribbrock, D., Göddeke, D., Gutwenger, C.: HONEI: A collection of libraries for numerical computations targeting multiple processor architectures. Computer Physics Communications 180(12), 2534–2543 (2009)
4. Falcou, J., Sérot, J., Chateau, T., Lapresté, J.T.: Quaff: efficient C++ design for parallel skeletons. Parallel Computing 32(7-8), 604–615 (2006)
5. ISO: ISO/IEC 14882:2003: Programming languages — C++. International Organization for Standardization, Geneva, Switzerland (2003), (§3.9)
6. Kirschenmann, W., Plagne, L., Ploix, S., Ponçot, A., Vialle, S.: Massively Parallel Solving of 3D Simplified P_N Equations on Graphic Processing Units. In: Proceedings of Mathematics, Computational Methods & Reactor Physics (May 2009)
7. Kirschenmann, W., Plagne, L., Vialle, S.: Multi-target C++ implementation of parallel skeletons. In: POOSC '09: Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing. ACM, New York, USA (2009)
8. Kirschenmann, W., Plagne, L., Vialle, S.: Parallel sp_n on multi-core cpus and many-core gpus. Transport Theory and Statistical Physics 39(2), 255–281 (2010)
9. McCalpin, J.D.: Memory Bandwidth and Machine Balance in Current High Performance Computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter pp. 19–25 (dec 1995)
10. NVIDIA: NVIDIA CUDA C Programming Guide 3.1 (2010)
11. Plagne, L., Ponçot, A.: Generic Programming for Deterministic Neutron Transport Codes. In: Proceedings of Mathematics and Computation, Supercomputing, Reactor Physics and Nuclear and Biological Applications. Palais des Papes, Avignon, France (September 2005)
12. Reinders, J.: Intel threading building blocks. O'Reilly & Associates, Inc., Sebastopol, CA, USA (2007)
13. Veldhuizen, T.L.: Arrays in Blitz++. In: ISCOPE '98: Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments. pp. 223–230. Springer-Verlag, London, UK (1998)